

A Logic Based Approach For Enforcing Coding Standards

*A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology*

by
G. Sri Ram

to the
**Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
March, 1998**

- 1 MAY 1998

Inv. No. A 125388

CSE-1998-M-SRI-LOG

Entered in system


Ans
4-5-98



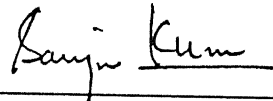
A125388

Certificate

Certified that the work contained in the thesis entitled "A Logic Based Approach For Enforcing Coding Standards", by Mr.G. Sri Ram, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.



(Dr. Harish Karnick)
Professor,
Dept. of CSE ,
IIT Kanpur.



(Dr. Sanjeev Kr. Aggarwal)
Associate Professor,
Dept. of CSE,
IIT Kanpur.

Abstract

This thesis explores techniques for developing tools for enforcing software coding standards. Source code which adheres to the principles of a well-established methodology is readable, maintainable and reusable. Coding standards are designed to incorporate such quality features in the developed source code. C++, a widely used language for object-oriented software construction, is a complex language with many pitfalls and therefore needs tools which help in enforcing coding standards. In this thesis, a logic based tool has been developed to check for coding standard violations. Coding standards are expressed as Prolog rules using a set of basic predicates which were arrived at after studying a large number of coding rules. This makes it possible to easily add new rules or remove existing rules. A public domain parser was used to convert the source program into abstract syntax trees. These trees were then traversed to assert facts about the source program in the Prolog database. The tool has been tested on programs deliberately seeded with violations as well as those developed by other students. The technique may be used for other languages also.

Acknowledgments

At the outset, I want to thank **Dr. Harish Karnick** for his support and guidance throughout my work. In spite of his various involvements he was always available and patient. I am also greatly indebted to **Dr. Sanjeev Kr. Aggarwal** for his guidance and support. I wish to thank my family for encouraging me to go for post-graduate studies.

I am grateful to Wipro (India) Pvt. Ltd. for providing a fellowship during my thesis. Apart from the ample financial support provided by the fellowship, it has been a source of inspiration to work in a field of industrial use.

My sincere thanks to the lab staff in the Department of Computer Science for their assistance and cooperation during this work. I would like to specifically thank Kiran, Kishore, Madhu, Atul and Rajesh who have helped during the course of this thesis work. All my classmates have also made my stay here a most memorable one.

Contents

1	Motivation	1
1.1	Scope of the Thesis	2
1.2	Organization of the Report	3
2	Coding Standards	4
2.1	Features of C++	4
2.1.1	C++ and OO programming	5
2.1.2	Problems with C++	5
2.2	Some Coding Rules for C++	6
2.3	Related Work	7
2.3.1	Initial Ideas	7
2.3.2	Current Tools	8
2.3.3	Other Code Analyzers	8
3	Design	9
3.1	Logic as a Specification Language	9
3.2	Overall Design	11
3.3	Prolog Facts for C++	12
3.3.1	Terminology	13
3.3.2	Basic Concepts of C++	14

3.3.3	Conceptual Schema	15
3.4	Comparison with other Approaches	17
3.5	Logic Programming in Software Engineering	17
4	Implementation	19
4.1	General Features	19
4.2	Implementation of PFGen	20
4.3	Primitives	24
4.3.1	Anatomy of a Coding Rule	24
4.3.2	SWI Prolog	25
5	Results	26
5.1	Choice of Rules	26
5.2	Test Suite	26
5.3	Third-party Programs	28
6	Conclusions	30
6.1	Synopsis	30
6.1.1	Limitations	31
6.2	Further Work	31
A	List of constraints for C++	32
B	Conceptual Schema	34

List of Figures

1	Overall Design of the C++ constraint checker	12
2	Structure of PFGen	21
3	Layout of the AST_LexYacc class	22
4	Inheritance hierarchy of the graph nodes	23
5	Prolog Specification of a Coding Rule	25

Chapter 1

Motivation

Software organizations either develop and market software products or create systems as per the requirements of their clients. Irrespective of its type, an organization inevitably ends up in modifying and reusing code. The product companies do this because they strive for upgraded versions of their products. For the others, this is due to the changing requirements of their clients. So, for a software company to thrive, it is necessary that the software produced by it meet high standards of quality in terms of robustness and maintainability, besides the specified functional requirements. This is the reason why organizations adopt some well established methodology in their software development activities. Each methodology has an underlying philosophy and designs which conform to it are generally successful. The next choice is that of a language which conveniently allows the design to be implemented. Ideally, the implementation language is consistent with the software methodology adopted. This may fail to happen due to the inadequate experience of the programmer with this language and/or methodology. Coding standards are designed to uncover such deficiencies from the source code. By themselves, coding standards will not benefit the organization. To be effective, they must be part of a comprehensive quality plan which operates during all phases of software development.

One such methodology is the object-oriented (OO) methodology and C++ is a very popular language which supports it. However, as the language does not enforce any particular style, C++ programs may not reflect design principles. Consider one

such principle which stipulates that *classes should not have any public data-members*. This translates into a coding constraint that no data-members be declared in the public interface of a class. As the compiler does not enforce such constraints either manual or automatic checking is necessary to test for conformance.

Manual checking of small one-time programs is convenient and simple. For large projects with interdependence among files, it is not only painfully time-consuming but susceptible to errors. Automated tools will be highly suitable for such mundane tasks. This forms the motivation for this work which develops a tool to check for conformance to coding standards.

1.1 Scope of the Thesis

Automated checking for uncovering violations of coding constraints involves analyzing the source code and then applying the rules to it. This implies that the rules are to be communicated to such a tool in some manner. Vendors differ in this aspect: some expect the source code to be annotated with commented rules whereas others follow the simpler route of incorporating a set of rules in their tool while a select few define a specification language to express coding rules.

A useful tool should not dictate a particular style but provide maximum flexibility to the user in terms of choosing a style. This is necessary because different organizations stipulate different coding rules. Even different groups in the same organization may need to differ on such rules. So, it is desirable that the tool be configurable to maximize its utility to the programming community. We were guided by this principle in developing our tool.

In this thesis, techniques which enable users to specify coding standards for C++ were investigated. A tool PFGen was developed which given a C++ program builds an intermediate representation of it. Coding rules written in Prolog are able to work on this representation to notify the undesirable constructs.

We mention the following assumptions that were made in developing the tool

- The input is a syntactically valid C++ program i.e., the input program must

have compiled without errors.

- Efficiency of such tools is not of primary concern. This is because code reviews are relatively infrequent as compared with compilations.

1.2 Organization of the Report

The rest of this thesis is organized as follows. In Chapter 2, we discuss the rationale behind the use of coding standards and present some such standards for C++. The design of our tool is discussed in detail in Chapter 3. How the idea of logic programming evolved in the context of coding standards is presented here. Chapter 4 is a discussion on the implementation aspects of the prototype C++ constraint enforcing tool. In Chapter 5, results of running the tool on some C++ programs are presented. Finally, we conclude with the limitations of this implementation and remarks on further work in Chapter 6.

Chapter 2

Coding Standards

The structured programming methodology led to the concepts of procedural abstraction. Popular and widely used implementations of these concepts are the C and FORTRAN language libraries which simplified coding to a large extent. Object-orientation aims to enhance reuse to more complete entities like data-structures, graphics packages etc. Principles like encapsulation, inheritance and polymorphism are the key features of this approach. This means that by using the principles of OO (object oriented) design it is possible to build class libraries which can then be used as components for other applications. This reduces the cost of design, coding and testing effort by amortizing it over several designs. Languages like C++ have been used to develop such libraries. But the collective experience of people over the years suggests that to gain the benefits of reuse careful planning and foresight must be invested. In the following discussion on coding rules, it is necessary to remember this. These rules are oriented towards making software reusable and easily maintainable. They are less relevant for simple one-time programs.

2.1 Features of C++

In this section, we review the features of C++ purely in relation to the OO approach. For detailed descriptions of the language and its features see [Str91], [Lip89].

2.1.1 C++ and OO programming

The C++ notion of class is the key to other OO features. The classes in the program model the objects in the application domain. Various relationships among classes may be represented by language constructs: *isa* relationship is represented by public inheritance, *hasa* relationship by membership. Generalization can be represented with a single base class with multiple specialized derived classes. In such cases, the generalized class can be made an abstract base. Since a C++ class is a type, relationships between classes receive support from compilers and are generally amenable to static analysis.

To reveal minimal amount of information access to data-members and implementation functions of a class can be reduced using `private` and `protected` specifiers. The `public/protected` distinction can be made use of to distinguish between the needs of designers of derived classes and general users.

2.1.2 Problems with C++

All is not well with programming in C++ . There are enough murky details to baffle an inexperienced programmer. Most of them are inherent in the language: having been part of the design goals.

One such aim was total C compatibility which ensured that all the evils of the C-language were preserved. For example, silent type conversions are a major source of problems in C. They continue to be so in C++ . The confusion is compounded by implicit conversions to user defined objects using single argument constructors and type conversion methods(defined for their classes).

Like its predecessor, C++ takes no responsibility for objects created on the heap. The programmer is required to keep track of the created objects and destroy them when they are no longer needed. This is tedious and is likely to lead to error-prone code.

C++ allows unrestricted access to memory locations using pointers. This may inadvertently lead to violations of encapsulation when pointers to private data-members are returned from access functions of a class.

Besides these, the language introduces enough idiosyncrasies of its own which may be occasionally forgotten even by experienced programmers. A striking example is that the semantics of assignment is different from that of initialization. Whereas for language defined primitive types this poses no problems, the implication for user-defined classes is that any of: constructor, copy constructor and overloaded assignment operator may be invoked depending on the context. Although the language clearly specifies which of them will be invoked, the rules are many and complex.

The following, from [R⁺91], aptly sums up the language:

C++ is a complex, malleable language characterized by a concern for the early detection of errors, various implementation choices and run-time efficiency at the expense of some design flexibility and simplicity.

2.2 Some Coding Rules for C++

Considerable discipline and expertise is needed to program effectively in C++. In this section, we discuss some of the coding rules that were implemented as part of this thesis. A complete listing may be found in Appendix A. These guidelines, borrowed from the work of C++ experts [Mey92], [Mey96], have been picked to form a representative set.

- **Declare an assignment operator and a copy constructor for each class declaring a pointer member.** The compiler declares a default copy constructor and assignment operator for a class if none is defined. The default action is bit-wise copying of members which, in the presence of pointer members, leads to multiple objects pointing to the same data.
- **Make destructors virtual in a base class.** Whenever a derived class object is deleted through a base class pointer, the destructor of the derived class is not called if the base class destructor was not declared virtual.
- **Avoid public data members.** This is purely in conflict with the style of programming advocated by an OO language. Besides the syntactic difficulties for users of the class, it means that the state of the object is no longer secure.

- **Use pass-by-reference-to-const instead of pass by value.** Passing an object by value is costly in terms of the number of function calls as well as usage of memory. These include constructors and destructors for the arguments etc. Passing a reference saves such overhead. However, it is possible to modify the object being passed through its reference. Use of `const` references is encouraged to prevent that.
- **Don't return handles to internal data structures unless they are secure.** A pointer or reference to private data returned from a public member function of class is a handle through which the state of the object can be modified. The handle is secure if the pointer/reference is a `const`.
- **Don't redefine an inherited non-virtual function.** A base class, if designed properly, is expected to declare those functions as virtual which are likely to be redefined for derived classes. When a non-virtual function is being redefined it is either the case that the base class did not predict (wrongly) such a use of the class or this derived class was not intended to be derived from the base class. In either case, the likelihood for error increases.
- **Don't overload `||`, `&&`.** The use of an overloaded operator is replaced with the equivalent function call. With C hovering in the background, a user of such an overloaded operator may expect short-circuited evaluation which obviously is not what the compiler provides.

2.3 Related Work

In this section, we trace the development of static analyzers from Lint to the ones currently offered by vendors. The work done in academic institutions is also discussed.

2.3.1 Initial Ideas

Lint is a popular tool that attempts to detect features of C programs that are likely to be error-prone, non-portable, or wasteful. It is generally available on all UNIX

systems. Similar ideas for C++ were first enunciated by Scott Meyers [ML91]. Their implementation of such a tool was called Clean++ [DMR92], which deviated from the lint routine of hard-coding rules in the tool. A metalanguage called CCEL was developed to allow users to express constraints.

2.3.2 Current Tools

Several vendors like ParaSoft Corp.[par], Gimpel Software [gim] offer tools which go the lint way. A notable exception is Abraxas Software [abr], whose tool named CodeCheck lets you specify in a C-like language what kinds of analyses are to be performed. Refer [MK97] for a survey of the relative merits of such tools.

2.3.3 Other Code Analyzers

Examples of systems that capture the structure and semantics of a program in a relational database are CIA [YNC90], CIA++ [GfC90] and REPRISE [RL91]. Such tools serve as a foundation for rapid development of programming tools. GENOA [Dev92] is a language independent application generator that can be used to generate a wide variety of code analysis tools.

Chapter 3

Design

In this chapter we describe how first-order logic can be made a specification language for coding rules. Rules written in such a language derive knowledge from a database of facts which incorporate the information of a C++ program. The conceptual schema for such facts is also explained. Next, we compare our method with the existing ones. Lastly, a brief survey describing the efficacy of logic programming for software engineering is presented.

3.1 Logic as a Specification Language

As pointed out earlier in Chapter 1, our goal was building a tool which is flexible and can be tailored to the needs of an organization. Such a tool requires that the rules are specified separately. To discover the structure of a possible language in which rules may be expressed, a large number of rules were collected, classified and analyzed. Our observations are:

1. The rules, framed in English, mostly have as nouns the ‘concepts’ of C++. These include language features like: *functions, classes, data-members, constructors, destructors* etc.
2. These nouns are qualified by adjectives which happen to be the attributes of the concepts. These include language terms like: *public, inline, virtual, static*

etc., and other generally understood features like *parent*, *global*, *references* etc.

3. The simplest rules assert that some attribute(s) of a concept hold. Though written in English in a manner more suitable for humans like:

Data-member is declared public.

they can be framed as :

public data-member.

to make explicit the ‘concept’ and ‘attribute’ features.

4. More complex rules compose the simplest ones with the operators **logical and**, **logical or**, **negation** and **implication**.

These observations led us to assume that a majority of the rules are the composition of the ‘attributed’ concepts of C++. However, the involvement of natural language precludes the possibility of having a tool which supports arbitrary rules. This is because alternative wordings for the same rule are possible. Hence, a tool can provide maximal support for addition of new rules but it is the responsibility of a human to strip a new rule of its natural language embellishments before formatting it in a manner acceptable to the tool. We do not consider this to be a serious limitation.

At this juncture we were left with two alternatives, the first of which was to design a language of our own which allows one to express such rules. The other was to use an existing language. We chose the latter.

We see that the rules can be easily expressed as well-formed formulae in first order logic. For example, the rule *make destructors virtual in a base class* is expressed as:

$\text{base_class}(C) \wedge \text{dtor}(Dt, C) \wedge \text{not_virtual}(Dt) \Rightarrow \text{bad_dtor}(C)$

and the rule *don't return handles to internal data-structures unless they are secure* is written as:

$\text{member_fct}(F) \wedge \text{returns}(F, Rt) \wedge \text{handle}(Rt) \Rightarrow \text{violates_encapsulation}(F)$.

Of course, the predicates *base_class*, *member_fct* etc., need to be defined for these formulations to be successful. All the collected rules were written in that notation and as can be seen, barring a few typographical rules, their specification was not only easy but also elegant. Our choice of using first-order logic was also influenced by the availability of a logic programming language, Prolog. The availability of such a

language is a blessing in more than one ways. The generality and power of an existing general purpose programming language based on logic can be gained without having to impose an absolutely new language on the users. This led to our choosing Prolog as the language in which the coding rules are to be expressed. We discuss next how the C++ program is converted into a Prolog facts database so that the Prolog interpreter becomes a checker for whether a given C++ program conforms to the rules in the Prolog database or not.

3.2 Overall Design

In the process of executing the specification of a rule Prolog needs access to the program information. The lowest level Prolog predicates can be made to invoke C routines which obtain the necessary information. Such routines extract the necessary information from a program database generated by a parser. However, we chose to collect the program information in a Prolog database. This ensures that all the pattern matching is now based on the unification of data-structures.

The design of our tool can be diagrammatically represented as shown in Figure 1. **PFGen** is the back-end of the tool which converts the given C++ program into Prolog facts. The structure of these facts is discussed later. For now it may be assumed that the entire **syntactic-semantic** information of the input program is preserved in Prolog as a set of facts. The left-hand side of the figure shows the 'black-box' which effects this conversion. Now, rules written in Prolog make use of an underlying set of *primitives* which extract information from the program dictionary. This set of primitives was built incrementally : i.e., additions were made as new rules were implemented and the process stabilized after a substantial set of primitives emerged. These include techniques and cliches found frequently in the rules. We believe that most new rules can be incorporated simply by using the existing primitives. However, it is possible that a radically new rule may necessitate the creation of new primitives.

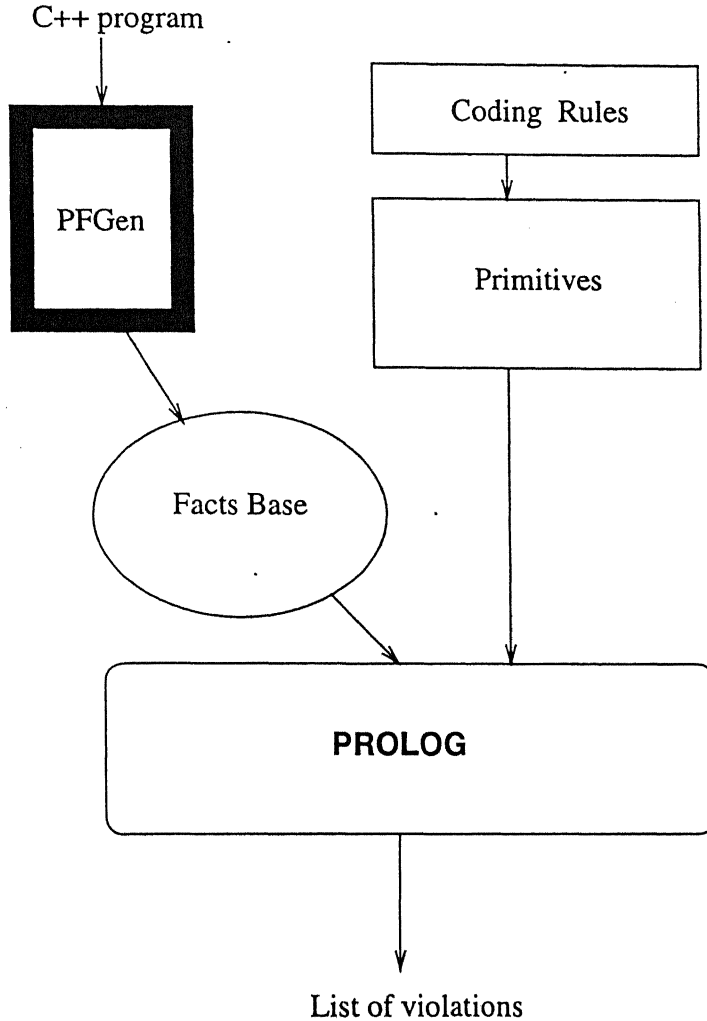


Figure 1: Overall Design of the C++ constraint checker

3.3 Prolog Facts for C++

In the previous section, it was assumed that the entire syntactic-semantic information of the C++ program was available as a Prolog database organized in a manner which makes it amenable to retrieval by logic programs. Before presenting the conceptual schema of such a knowledge-base, the terminology of logic programming [Das92], [SS94] used in this context and the basic concepts of the C++ language are reviewed.

3.3.1 Terminology

A **fact** is a means of stating that a relation holds between objects. An example is,

`obj_var(alpha, "first.cc").`

This fact says that `alpha` is a variable declared in the file `first.cc`. A set of facts is a description of a situation.

Queries are the means of retrieving information from a logic program. A query is similar in form to a fact. However, in place of **constants** (such as `alpha` in the above example) it has **variables** which stand for unspecified entities. An example is,

`obj_var(X, "first.cc")?`

Answering a query with respect to a program is determining whether the query is a logical consequence of the program. Logical consequences are obtained by applying deduction rules. A query containing a variable (such as `X`) asks whether there is a value for the variable that makes the query a logical consequence of the program. The answer to the above query is $\{X=\text{alpha}\}$.

The **term** is the single data-structure in logic programs. It is defined inductively. All constants and variables are terms. **Compound terms** are of the form

$f(t_1, t_2, \dots, t_n)$

where f is called the *principal functor* and each of the t_i 's is a term. The **arity** of this term is n . In the above example, `obj_var` corresponds to f , `alpha` to t_1 , `"first.cc"` to t_2 . Terms in which variables do not occur are called **ground** whereas those in which variables occur are called **non-ground**. Note that all facts which we generate for a C++ program are ground clauses.

The entity `obj_var(...)` without a succeeding punctuation mark i.e., a question mark or a period, is called a **goal**. The most interesting feature of logic programming, **rules** are statements of the form,

$A \leftarrow B_1, B_2, \dots, B_n.$

where $n \geq 0$ and A and all the B_i 's are goals. A is called the **head** of the rule and all the B_i 's, forming a conjunction of goals, make up the **body** of the rule. Rules are a means of defining new or complex relationships using other simpler relationships. The rule is read as:

“ if B_1 and B_2 and ... B_n then A ”.

An example of a coding rule in this notation is:

```
bad_dtor(C) :- base_class(C), dtor(Dt,C), not_virtual(Dt).
```

3.3.2 Basic Concepts of C++

A C++ program consists of declarations, perhaps encapsulated in classes or functions and statements within functions. A declaration is a binding of attributes to a name. These attributes include *scope*, *type*, *storage* and *linkage*. A name can be used only within a region of program text called its *scope*. The four principal scopes are: *local*, *function*, *file* and *class*. Each name has a *type* which determines its use. Types may either be primitive (like *int*, *short*, *float* etc.) or derived. Derived types are constructed from the fundamental types by: *arrays*, *functions*, *pointers*, *references*, *constants*, *classes*, *structures*, *unions* and *pointers to class members*. A named object has a *storage class* that determines its lifetime. In C++ a declaration may specify the storage class as either *auto* or *static*.

The following from the *Annotated Reference Manual* (ARM) [ES90] succinctly describes all the statements provided in this language.

C++ provides statements for conditional execution (*if* and *switch*) and iteration (*while*, *do* and *for*). The *break*, *continue*, *return* and *goto* transfer control within the program. Other statements evaluate an expression or may do nothing (null statement). Statements may be grouped together to form compound statements.

For each statement and declaration an additional attribute, *location* in file, is necessary to generate helpful messages for the user.

Expressions, the primary building blocks for computation, are yet another important feature. The language provides a number of operators to construct expressions. The usual arithmetic and relational operators, operators for conditional evaluation, storage management, bit manipulation, pointer manipulation etc.. It also provides the ability to redefine the meaning of most C++ operators when at least one operand is a class object.

The ARM is the ideal place to look for a complete understanding of the features of the language. The basic concepts were presented here, only for the sake of continuity and completeness.

3.3.3 Conceptual Schema

The background of the last two subsections is necessary to understand the conceptual schema of the Prolog facts for C++. The design of the conceptual schema is presented in three phases. The first of these specifies the schema for declarations, the next for statements and the last for expressions. As details may hinder understanding, the examples given in this section are all stripped-down versions of the actually implemented Prolog facts. Refer Appendix B for the exhaustive schema design.

A simple declaration like

```
int ivar;
```

is translated to

```
obj_var(scope(2), primitive(int), storage(global),  
        name("ivar"), location(5)).
```

The *principal functor* for all declarations is `obj_var`. The *scope* attribute has an integer value indicating the scope in which this variable was declared. All the scopes that occur in the program are numbered and these are used in disambiguating the use of a *name*. The *type* attribute in the above example happens to be primitive. An example of a more complex type declaration and its representation in the knowledge-base is

```
char (*( *YYLVAL())[3])();
```

```
type(function_returning (  
    pointer_to (  
        array_of ( 3,  
                    pointer_to ( function_returning (char) ))  
                )))
```

The reading: `YYLVAL` is a function returning a pointer to an array of pointers to functions returning `char` (both the functions do not take any arguments) shows how close it is to a natural language description of the declaration.

For a class declaration like,

```
class Student: public Person {  
    // ...  
};
```

the significant attributes are the scope which it defines, the list of base classes and the other information like: its name, scope in which it has been declared and location in the file.

```
classdef(name("Student"),  
    scope(2),  
    scope(11), // scope defined by this class  
    [basespec(accs(public), accs(nonvirtual),  
        typeclass(name("Person"), scope(2)))],  
    location(28)).
```

Declarations of *structs* and *unions* have the same attributes except that the principal functor is named as `structdef` or `uniondef` respectively.

The principal functors for statements are `expr_stmt`, `ret_stmt`, `for_stmt` and so on. The terms comprising the attributes of a statement include its scope and the computation defined by it in the form of expressions and references to other statements. Every statement has as its first attribute the scope of the block in which it occurs.

Expressions do not appear as principal functors. They appear within statements. However, they are very naturally expressed in *reverse Polish* notation. An example may help clarify this,

```
x = y + z;  
  
expr(opr(=), name("x"), expr(opr(+), name(y), name(z)))
```


3.4 Comparison with other Approaches

Providing the ability to add new rules to the system has eliminated all comparisons with tools that do not include that facility. It is more interesting to compare the different configurable tools which differ on the specification language used. CodeCheck [abr], an industry-level tool, uses C as the specification language whereas CCEL [DMR92] is a metalanguage which is very similar to C++. Someone who enunciates coding rules for C++ is likely to be an expert conditioned to thinking in the language. So, writing rules in a C++-like language will be natural for such a person. However, if we consider such rules as a repository of wisdom to be made available to novices then someone reading those rules is going to face the same problems as with reading C++ code. On the contrary, Prolog rules are understood more easily as they are closer to natural language. Developing a similar tool for a different language takes less effort with this approach. Language parsers and Prolog, the required components are available as public domain software and designing the facts base is the only thing to be done. But we do not propose the Prolog interface for practicing programmers. A user-friendly interface independent of the language for which constraint checking is being done has to be developed.

3.5 Logic Programming in Software Engineering

This section is intended to serve as a pointer to the vast amount of research in this field. The focus is on the various uses that others have found for logic programming in software engineering.

One of the significant reasons for the use of logic is the availability of a flexible and efficient implementation, Prolog. It is flexible because it may be used as a rule-language, as a relational data modeling language or as an executable specification language.

The applications may be classified with respect to the environment in which they are used [CL93]

- **Programming in the small:** Compilers and debuggers were classified in this

category. One particularly interesting project: CENTAUR uses logic programming technology for building a structure editor based on Prolog. A structure editor is a tool that uses abstract syntax to perform a number of semantic analyses on the program being developed.

- **Programming in the large:** Tools which aid in intelligent retrieval of data from programs and in version and configuration management were put under this heading. Of interest to us are reverse engineering tools like [GAU92] which introduced Prolog for prototyping solutions for reverse engineering problems.
- **Programming in the many:** Developing tools for *process programming* have been listed in this category. These tools are required to co-ordinate the efforts of different agents : human or otherwise during the software development process.

Chapter 4

Implementation

The design described in Chapter 3 indicates that two distinct modules: **PFGen** and the library of primitives need to be implemented.

4.1 General Features

PFGen converts the given C++ program into a set of Prolog assertions and ensures that all the syntactic-semantic information of the program is preserved therein. The general strategy [Hol95] to achieve this is to construct the syntax tree and generate the facts while traversing it top-down. Top-down traversal facilitates the generation of facts because the nodes representing high-level constructs are visited before the nodes which represent their attributes. Unlike C, C++ does not have a simple Yacc-acceptable grammar and writing a parser requires quite an effort. As C++ parsers are available in the public domain we avoided replicating existing software. `cppp` developed at Brown University which generates an ASCII representation of an abstract syntax tree from the given C++ program was selected for this purpose. As suggested in the documentation provided with `cppp`, the abstract tree is parsed to reconstruct the structure which is then traversed. Although, conceptually `cppp` produces an abstract syntax tree the actual structure happens to be a graph because of the additional links to reflect semantic information such as linking a use to a declaration.

At the Prolog end, a set of primitives were arrived at after implementing a variety

of rules. We converged on a set of basic predicates by expressing rules of various types in the form of Prolog rules. Variety, in the rules, was sought for to maximize the set of primitives.

4.2 Implementation of PFGen

The generation of facts is done in two phases: *i)* constructing the parse tree corresponding to the given program and *ii)* traversing it to generate the facts. We parse the textual representation of the graph produced by ccpp using the LALR(1) parser generator, Yacc and simultaneously construct an in-memory version of the graph. Having chosen C++ as the implementation language the nodes can be represented as objects. As the graph contains nodes of different types, organizing the corresponding classes in a hierarchy is natural. By associating specific procedures or behaviours with the various classes of graph nodes, generation of facts reduces to a depth-first traversal of the graph punctuated with calls to the function *emit_pf*. The basic structure of PFGen is shown in Figure 2.

The class which manages the entire show by co-ordinating the creation of objects and the timings of their various activities is *AST_Driver* (shown as controller in the figure). The *AST_LexYacc* is the class which encapsulates the behaviour of the scanner and the parser. The coupling between the two sub-systems is tight enough that they have been modeled as a single class. The class *AST_Term* encapsulates all the important information associated with the various valid lexemes encountered by the scanner. The class *GNode* is the base class for all the nodes which appear in the graph. It has been made an abstract base class by defining *emit_pf* as a pure virtual function.

The class *AST_Driver* processes the command line arguments and instantiates an *AST_LexYacc* object with the input file as a constructor argument. This object parses the file and on a successful parse returns a pointer to the head of the graph. The *AST_Driver* invokes the traversal function on the head of the graph which recursively invokes it on its child nodes.

The *AST_LexYacc* class encapsulates the scanner and parser as its public member

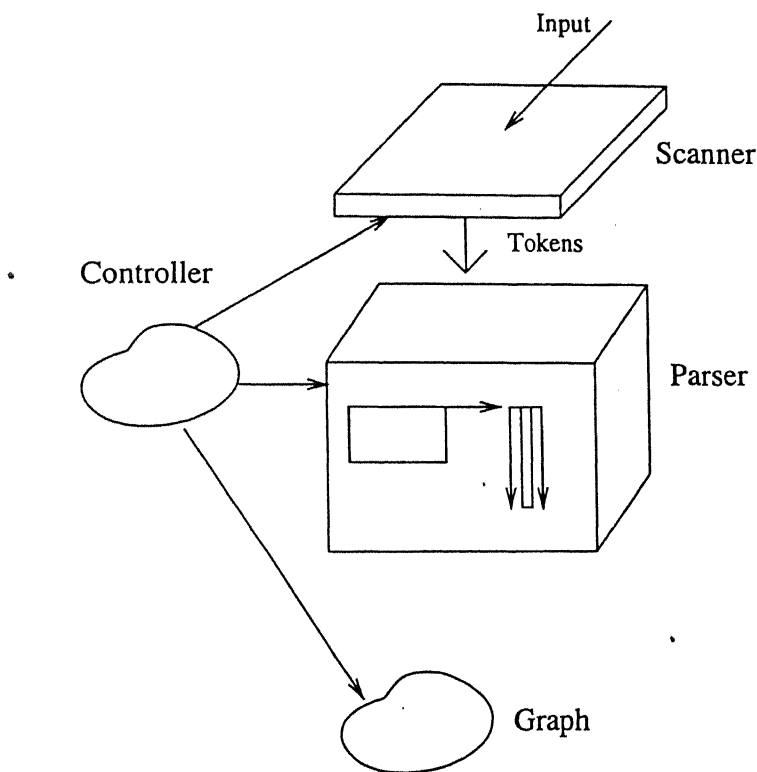


Figure 2: Structure of PFGen

functions. The parser was automatically generated using Yacc. The layout of this module is shown in Figure 3.

The class *AST_Term* is relatively simpler. It encapsulates all the information associated with a token. This includes the name of the node, any associated line number and file name alongwith some control information generated by cppp.

The classes whose objects appear as nodes of the graph are organized as a hierarchy. This not only facilitates their specification but also makes explicit their commonalities. Although there are a large number of such classes they can be classified as:

- Objects
- Statements
- Expressions

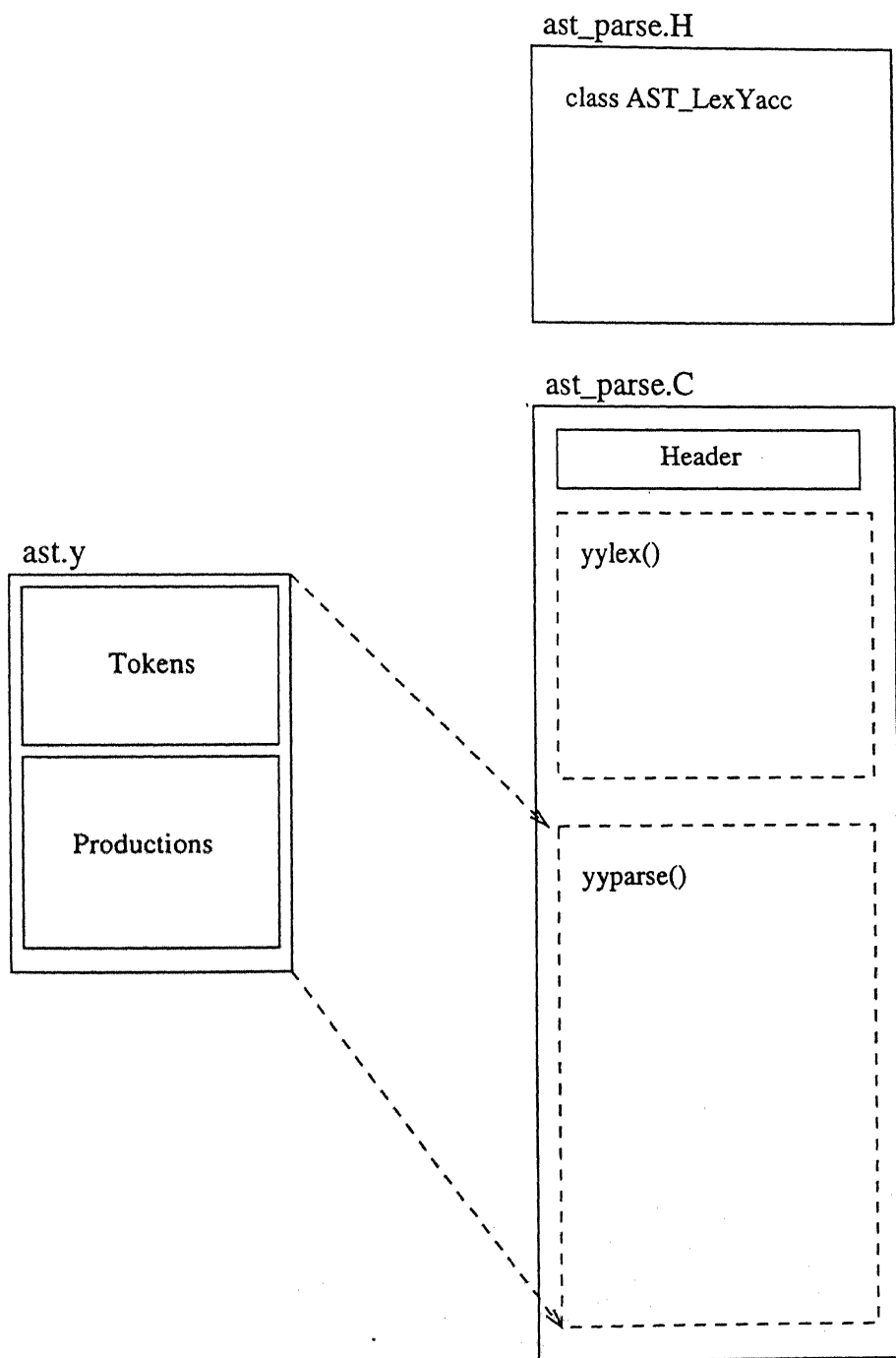


Figure 3: Layout of the `AST_LexYacc` class

- Types
- Lists
- Others

The first category includes most abstracts objects that one finds in C++ source code such as variables, classes, functions, typedefs, enums. The respective classes are *Obj_Variable*, *Obj_Class* etc. The next three categories describe, as the names suggest, statements, expressions and types. Lists are necessary in various contexts, the most common being function argument lists and list of variables declared in a scope.

So, the overall class hierarchy consists of these six classes, each of which derives from *GNode*. The classes, whose objects actually appear on the graph, derive from one of these. A partial view of the inheritance graph is shown in Figure 4. Classes which are base classes of some other class are shown with a dark outline.

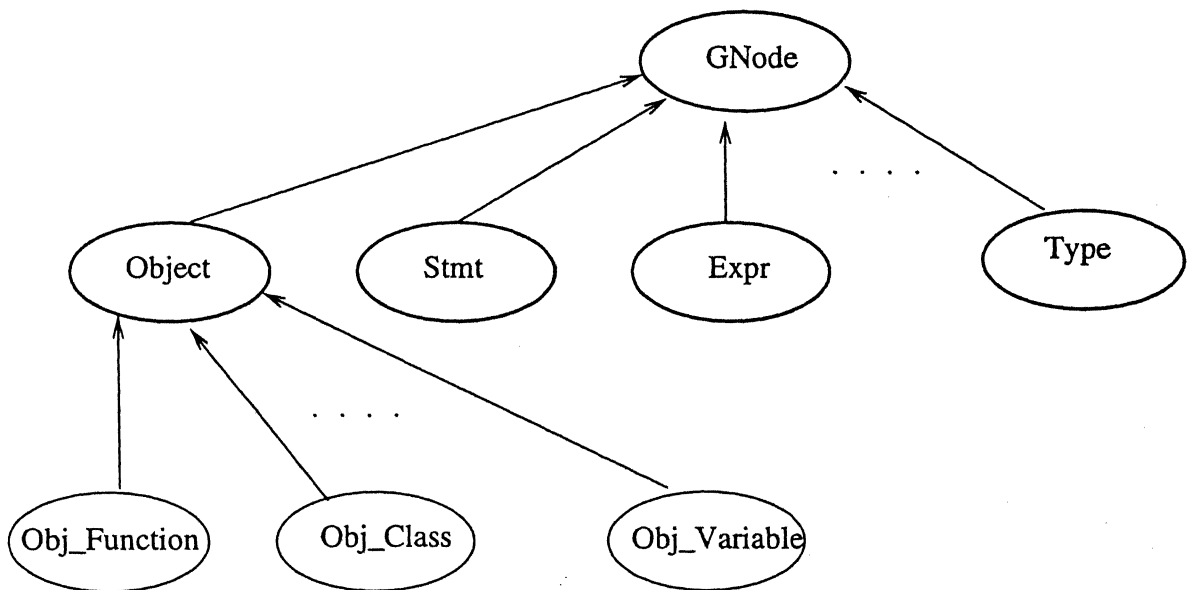


Figure 4: Inheritance hierarchy of the graph nodes

PFGen required the use of two data-structures: linked lists of the class *GraphNode*

and maps for pairs of strings and *GraphNode*s. We chose not implement these and used their template versions from the GNU class-library.

4.3 Primitives

The module **PFGen** populates the facts base as described above. From Chapter 3 the schema of those facts is also known. This forms the underlying repository of program information to be used by the rules. Rules for coding standards are specified in Prolog and are executed by the Prolog interpreter as queries. Refer [CC87] for an interesting discussion of the Prolog interpreter. In this section the specification of the rules is explained and it is interesting to see how this specification leads to a set of primitives which occur repeatedly in the rules.

4.3.1 Anatomy of a Coding Rule

Consider the rule shown in Figure 5 and its specification in Prolog. The inference mechanism of Prolog executes this rule as a query and a violation is indicated if it is a logical consequence of the facts asserted in the database. Delving into the formation of the rule, we find that it is a *conjunction* of three sub-goals, the last being a message to the user. The Prolog interpreter attempts to satisfy the sub-goals beginning from the leftmost i.e., it picks `clscope` as the first one, to satisfy. Upon, success, the variables `S` and `C1` are instantiated and Prolog attempts to satisfy the second sub-goal i.e., `obj_var` with this value of `S`. If it encounters success, the third sub-goal `writeln` is tried which succeeds because such imperative goals never fail. Success of all the sub-goals leads to the success of the rule head. Note that in the attempt to satisfy a sub-goal (say, `clscope`) new predicates (like, `classdef`) may have to be satisfied recursively. Failure at any sub-goal leads to backtracking. If a goal is not a logical consequence of the stored facts, it never succeeds i.e., in spite of backtracking. With this, rather naive, understanding of the Prolog inference system, the coding rules given in Appendix A can be specified provided primitives are available. In the given example, `clscope`, `obj_var` are the primitives. This set of primitives is written in Prolog. It is not a finished subsystem but a kernel on which the user may build a

personal set as and when necessary. It is this feature which gives the tool the ability to handle new rules. Since predicting the primitives *a priori* is, in general, difficult, an incremental approach was taken to arrive at a reasonably comprehensive set.

Rule :- *No public data members to be declared in a class.*

```
faulty_cldata :- clscope(S, C1),

                obj_var(scope(S), _, prot(public), _, _, _,
                        name(Nm), location(Loc)),

                writef(" Line %t: '%d' is a public data member of
                        class %d \n", [Loc, Nm, C1]).

/* :a class scope S with class name T is searched for */
clscope(S,T) :- classdef(name(T), _, scope(S), _, _).
```

Figure 5: Prolog Specification of a Coding Rule

In the above rule, the first sub-goal selects the facts corresponding to *classes* in the input C++ program. Many coding rules require such sub-goals albeit with slight variations. One such primitive is `fct(F, scope(S))` which selects *functions*. Other interesting primitives are:

```
memberFunction(F); // select a member function
arg_list(F, Lf);   // retrieve the argument list of a function
fct(F, S, "operator new"); // select the named function
```

4.3.2 SWI Prolog

The primitives were implemented on a version of SWI-Prolog(v1.9.5) which runs on the Linux operating system. A freely available implementation of Prolog, distributed by the University of Amsterdam, it has a rich set of built-in predicates, a reasonably good performance and debugging facilities.

CENTRAL LIBRARY
I I T KANPUR

Acc. No. A 125288

Chapter 5

Results

Programs seeded with violations, of rules mentioned in Appendix A, were used to test the tool. Besides these, programs written by others were also used. Such programs capture, atleast partially, the real application of a constraint checker.

5.1 Choice of Rules

The object oriented paradigm emphasizes the specification of **data** in the program, regarding both the structure and the operations which access and modify it. C++ supports such concepts with the language feature `class`. Thus, programs are encouraged to structure data and make explicit their relationships in declarations. The coding rules in Appendix A are intended to eliminate common mistakes from declarations and this choice is in tune with the aims of the language.

5.2 Test Suite

Once the rules were fixed upon, programs that violated the rules were written to test the developed tool. Small programs, typically less than twenty lines, were used to test the rules individually i.e., each such program had violations of a single rule. After debugging the Prolog specification of a rule on such a program, it was tested on a

set of seven programs. These had an average size of seventy lines and had fragments from real C++ programs. The following fragments, taken from the test programs, give a flavour of some of the common mistakes in declarations.

```
class Assorted {
    int*  ilist;
    char* clist;
    short tag;

    public:
        Assorted(int i, char c, short t);
        int* get_ilist() const;
        char* get_clist() const;
        short get_tag() const;
};
```

The program containing this fragment, as all other test programs, is compiled even without a warning by the GCC g++ compiler with all warnings turned on (compiler option `-Wall`). However, good coding practices stipulate that the access functions `get_ilist`, `get_clist` not return handles to internal data, thereby throwing making possible external manipulation of class data. The fragment elicits the following message from the tool:

```
Line 30:  Returning a handle to a data member less accessible
          than the function from 'get_ilist'
Line 31:  Returning a handle to a data member less accessible
          than the function from 'get_clist'
```

Similarly, the piece of code

```
class Base {
    // ...
    virtual void print() const;
};

void write_obj(const Base b)
{
    b.print();
}
```

is *legally* valid but is poor programming. Not only is passing an object by value costly in terms of number of function calls but a possible use of this function is:

```
class Derived : public Base {  
    // ...  
    void print() const ;  
};  
Base b;  
Derived d;  
write_obj(b); // works fine  
write_obj(d);
```

To a function taking a passed-by-value, instead of a passed-by-reference, base class parameter p , the specialized behaviour exhibited by the derived class object p which has been passed is “sliced off” and p behaves like a base class object, even when virtual functions are called on it. Hence, in the given example, only the base class version of *print* is invoked in the second call to *write_obj*, which may not be as expected. The tool indicates the offending construct as

Line 64: Passing object by value

The performance of the tool on these test programs was as expected. This is because such programs were hardly beyond hundred lines. But, it was gratifying to note that the tool detected inadvertent violations in the benchmark programs. Also, for each violation that was seeded, constructs which were close but not violations were also made part of the program. A few stray violations were observed for some such constructs. For instance, the coding rule *an overloaded assignment operator takes a constant reference to the class as its argument* results in the correct message if an assignment operator function fails to satisfy the constraint, but does not warn if an assignment operator was not declared.

5.3 Third-party Programs

As **PFGen** was written in C++ it was used to test the tool. The file which contains the declarations of all possible graph nodes was used for this purpose. Some of the messages, which are self-explanatory, are

Line 29: 'hexnum' is a public data member of class AST_Term
Line 52: Handle to data member being returned from constant
member function get_filename of class GNode
Line 69: Handle to data member being returned from constant
member function operator [] of class AST_Leaves
Functions declared implicitly in class "IfStmt": operator construct
Functions declared implicitly in class "SwitchStmt": operator =
No virtual destructor in base class Stmt
No virtual destructor in base class Data
No virtual destructor in base class Expr

Programs written by others were also used to test the tool. For this, we gathered programs written by other persons belonging to the same community as ours. Such programs can be classified as: written by those learning C++ and written as part of assignments. The programs of learners tended to use only the C-part of the language. Therefore, not many violations were observed except for *functions defined implicitly by C++*. The assignment programs generally make use of header files for graphics, threads etc., which use non-C++ constructs causing the parser to abort. For those, which do not make use of such constructs, the violations consisted mostly of *functions defined implicitly by C++*, *non-virtual or no destructor in base class*. There were hardly any violations arising out of *members redefined in derived class* etc., because inheritance was a less used feature.

Moreover, with large programs the parser output runs into thousands of lines and the system, at times, truncates such large files. This indicates that PFGGen should call the parser as a function and work upon the in-memory version of the tree to generate the facts. Also, with a more robust C++ parser, it is likely that the problems with header files are absent.

Chapter 6

Conclusions

We conclude with a synopsis of the scope and extent of this thesis. Some observations for carrying this work further are also made.

6.1 Synopsis

Our motive was to work out techniques which would lead to the development of a configurable tool for enforcing coding standards in C++ programs. We were instantly confronted with the problem of specifying coding standards in a manner suitable for a tool which weeds out the violations. Searching for a specification language led to an exhaustive analysis of the rules collected by us. Observing the features of those rules, we selected first-order logic as the specification language. This necessitated that the C++ program information be stored in a form amenable to such a language. Hence, the conceptual schema of a knowledge base for storing C++ program information was designed.

Having worked out an approach for specifying the rules and the schema of the database on which they operate, we set out to implement a prototype tool. The output of a publicly available C++ parser was made use of to generate the knowledge base in Prolog. Primitives, to be used as the basic building blocks, were developed for a substantial set of rules. As the primitives were developed incrementally, we noticed that the last few rules were the easiest to add. This was because they had the

advantage of using primitives developed earlier. We interpret this as an advantage for this approach.

6.1.1 Limitations

The most striking shortcoming of the prototype is the lack of a user-friendly interface without which a tool is doomed to ephemeral existence. However, adding a user-interface is customary as the process is well understood. More serious limitations stem from the use of the publicly available C++ parser. As the current version of this parser does not support templates the developed tool is also incapable of handling them. Other additions to C++ like RTTI and name spaces are obviously not present. Besides this, the use of the output of this C++ parser calls for more rigorous testing of the developed tool because of the possibility of undocumented features cropping up.

6.2 Further Work

The effectiveness of a tool lies in its being comprehensive and configurable. Rules for data-flow, inter-procedural and otherwise, have to be implemented to make the C++ constraint checker comprehensive. Besides, as mentioned earlier, an easy to use interface is to be added to make the tool complete.

Organizations typically use more than one language in their software development activities. A tool which supports multiple languages would be of use to them. Moreover, the design of constraint checkers for different languages will be similar. The language grammar, the database schema and the primitives have to be worked out but the underlying framework remains essentially the same.

Appendix A

List of constraints for C++

As mentioned earlier, the coding rules given here have been collected from various sources. They have resulted mostly from the experiences of others and can be profitably used by novices and practitioners alike. The rationale for such rules is found elsewhere, notably [Mey92], [Mey96], [gim]. These were the rules that were implemented in the prototype developed.

Memory Management

Rule: *Avoid hiding the default signature for operator **new** and operator **delete**.*

Rule: *Write **delete** if you have to write **new**.*

Classes

Rule: *Avoid public data members.*

Rule: *Avoid class data members whose type is reference.*

Rule: *Classes with dynamically allocated members in constructors ought to have defined a copy constructor and an assignment operator.*

Rule: *The sole argument to the copy constructor must be a **constant** reference to*

the class.

Rule: *Know the functions implicitly declared for a class by C++.*

Rule: *Do not declare constructors private.*

Rule: *Pass (and return) objects by reference instead of by value.*

Encapsulation Violations

Rule: *No handles to data members to be returned from **constant** member functions.*

Overloading

Rule: *Overloading the operator `||`, `&&` and `,` operators is not recommended.*

Rule: *Make the overloaded assignment operator function return a **constant** reference to the class.*

Rule: *Avoid defining type-conversion for user-defined classes.*

Rule: *Avoid overloading on pointer and numerical types.*

Inheritance and Virtual Functions

Rule: *Do not hide non-virtual members of public base classes.*

Rule: *Avoid diamond-shaped hierarchies.*

Rule: *Avoid functions in derived class with a name same as that of a base class function but with a different signature.*

Rule: *Base classes should declare a virtual destructor.*

Appendix B

Conceptual Schema

The design of the conceptual schema is central to the understanding of this tool. The schema of most of the important facts which are generated by **PFGen** are presented here. The following is the format of each entry: first, the schematic representation of the fact is given alongwith explanatory comments for some of the **terms**. An example of such a fact and the program fragment corresponding to it are also included.

Scopes

There are four kinds of scope: *file*, *local*, *function* and *class*. A name declared outside all blocks and classes has file scope and is said to be *global*. In the schema such a name has the attribute `scope(2)`. Names declared in other scopes have a different number (≥ 10) as their scope attribute. It is neither necessary nor ensured that these numbers follow an order which is in tandem with the program order; only uniqueness of these numbers is guaranteed. Names of formal arguments for a function are treated as if they were declared in the outermost block of that function.

Declarations

Class, Struct and Union

```
classdef(name(),  
    scope(),          // in which this class was declared  
    scope(),          // generated by this class  
    [],              // the list of bases  
    location()).
```

The principal functor for `struct` and `union` are `structdef` and `uniondef` respectively. The schema for each element in the list of bases is as:

```
basespec(accs(),      // access specifier  
    accs(),           // virtual or non-virtual inheritance  
    typeid()).        // the name of the base class .
```

Example

```
class Student : public Person {  
    // ...  
};  
  
classdef(name("Student"), scope(2), scope(11),  
    [basespec(accs(public), accs(nonvirtual),  
    typeid(name("Person"), scope(2)))], location(25)).
```

Functions

```
obj_fct(scope(),      // in which this function was declared  
    fct_ret(),        // complete type specification  
    prot(),           // protection specifier  
    link(),           // linkage specifier  
    stor(),           // storage specifier
```

```

kds(),
[],          // the default initializer list
inline,      // or not_inline
virtual,     // or not_virtual
name(),      // the name of the function
location()).

```

The term `mfct_ret` appears in the type specification for member functions. The term `kds` has the attribute `undefined` if the function has no definition. Otherwise, the scope introduced by the function body appears at this position.

Example

```

int main(){

obj_fct(scope(2), fct_ret(prim(int), []), prot(public),
link("C++"), stor(gfunction), scope(21), [], kds(not_inline),
kds(not_virtual), name("main"), location(69)).

```

Variables

```

obj_var(scope(),          // in which this variable was declared
type(),                  // type of the variable
prot(),                  // protection specifier
link(),                  // linkage specifier
stor(),                  // storage specifier
init,                    // initializer expression
name(),                  // the name of the variable
location()).

```

Example

```

Stack sArray[10];

```

```
obj_var(scope(21), array_of(typeclass(name("Stack"), scope(2)),
integer(10)), prot(public), link("C++"), stor(auto), no_init,
name("sArray"), location(72)).
```

Inherited Variables

```
obj_inh(name(),           // name of the inherited variable
        scope(),         // of the base class
        scope(),         // of the derived class
        prot(),          // current protection specifier
        access()).       // virtual or non-virtual inheritance
```

Example

```
obj_inh(name("average"), scope(12), scope(11), prot(protected),
accs(nonvirtual)).
```

Typedefs

```
obj_typedef(scope(),
            type(),           // type being renamed
            prot(),
            link(),
            name(),          // the type name being introduced
            location()).
```

Example

```
typedef unsigned long size_t;

obj_typedef(scope(2), prim("unsigned long"), prot(public),
link("C++"), name("size_t"), location(102)).
```

Enums

```
obj_enum(name(),
         type(),
         prot(),
         link(),
         name(),
         location()).
```

Example

```
enum xyz { X, Y, Z};

obj_enum(scope(2), enum(name("xyz")), prot(none), link(C++),
name("xyz"), location(10)).
```

Statements

Compound Statement

```
cmpd_stmt(scope(),           scope in which it occurs
                        scope(),       scope defined by this compound statement
                        location()).
```

This fact finds use in making explicit the compound statement introduced by `for`, `while`, `do` etc., statements.

Example

```
cmpd_stmt(scope(11), scope(15), location(9)).
```

Block Statements: Iteration

```
while_stmt(scope(),           // scope in which it occurs
```

```

    expr(),           // the expression upon which iteration depends
    scope(),          // scope defined by the while statement
    location()).

```

The schematic representation for do and for statements follows similar lines. The principal functors are `do_stmt` and `for_stmt` respectively.

Example

```

while(b < d) {

while_stmt(scope(11), expr(
opr("operator <"), name("b", scope(11)), name("d", scope(11))),
scope(14), location(19)).

```

Block Statements: Selection

```

if_stmt(scope(),      // scope in which it occurs
    expr(),           // the expression upon which control flow depends
    scope(),          // scope defined by the if statement
    location()).

```

Example

```

if (d != 0) {

if_stmt(scope(11), expr(opr("operator !="), name("d", scope(11)),
integer(0)), scope(15), location(15)).

```

The schematic representation for if-else and switch statements follows similar lines. The principal functors are `if_else` and `switch_stmt` respectively.

Expressions

Binary and Unary Expressions

```
expr(opr(),           // the operator in this expression
     expr(),
     expr()).
```

The only difference in a unary expression is that instead of two `expr`'s only one expression appears alongwith the operator. The operator could be one of the forty-four valid C++ operators with the correct *arity*.

Example

```
e = c - d;

expr(opr("operator ="), name("e", scope(11)),
expr(opr("operator -"),
name("c", scope(11)), name("d", scope(11))))
```

Other Expressions

Most other expressions like `delete`, `index_expr`, `if_then_expr` are, as their names suggest, specific expressions with the corresponding operators. Similarly, `callexpr` has the expression corresponding to the function name as its first argument and the list of actual parameters as the second argument.

Types

Primitive Types

```
prim(...).
```

Any of the C++ supported primitive types like `int`, `short`, `unsigned int`, `long`, `char`, `float` etc. may appear as the constant for the term `prim`.

Pointers, References, Arrays and Functions

```
pointer_to(type()).
```

```
reference_to(type()).
```

```
array_of(type(),
```

```
    integer()).    // the size of the array
```

```
fct_ret(type(),    // the return type
```

```
    []).          // the list of argument types
```

These are the most frequently used of the derived types. They are recursive and hence at least one of their arguments is `type`.

Example

```
char** argv;
```

```
int letters[26];
```

```
void update(int* val);
```

```
pointer_to(pointer_to(prim(char)))
```

```
array_of(prim(int), integer(26))
```

```
fct_ret(prim(void), [pointer_to(prim(int))])
```

Qualified Types

```
typequal(type(),
```

```
    kds()).    // maybe kds(const) or kds(volatile)
```

Types may have either the `const` or `volatile` qualifier or both. The example given is of the `this` pointer which is an implicit argument to all member functions.

Example

```

T *const this;
typeid(pointer_to(typeclass(name("T"), scope(2))),
kds(const))

```

Miscellaneous

Protection, Linkage and Storage Specifiers

The terms `prot`, `link` and `stor` represent the protection, linkage and storage specifiers of any declared entity. The possible values for these are

```

prot    none, public, private, no_access
link    C++ , C
stor    undefined, sfunction, gfunction, efunction, auto, register
        static, extern, argument, field, global, temporary , junk .

```

Constructor Initialization List

```

ctor_init(scope(),           // scope of the constructor definition
          ctor_init(), ...   // as many as there are expressions

```

Constructors have initialization lists in which each initializer is of the form: a class member or base class followed by an expression.

Example

```

Toy::Toy(int fir, int sec) : t1(fir), s1(sec) { }

ctor_init(scope(12), ctor_init(name("t1"), [name("fir")]),
          ctor_init(name("s1"), [ name("sec")])).

```

Bibliography

- [abr] Abraxas Software. URL: <http://www.abxsoft.com>.
- [CC87] W.F. Clocksin and C.S.Mellish. *Programming in Prolog*. Springer-Verlag, 1987.
- [CL93] Paolo Ciancarini and Giorgio Levi. What is Logic Programming Good for in Software Engineering? Technical Report UBLCS-93-9, University of Bologna, 1993.
- [Das92] Subrata Kumar Das. *Deductive Databases and Logic Programming*. Addison-Wesley Publishing Co., 1992.
- [Dev92] Premkumar T. Devanbu. GENOA- A customizable, language and front-end independent code analyzer. In *Proceedings of the International Conference on Software Engineering*, 1992.
- [DMR92] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A Metalanguage for C++. Technical Report CS-92-51, Brown University, 1992.
- [ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, USA, 1990.
- [GAU92] G.Canfora, A.Cimitile, and U.deCarlini. A Logic-Based Approach to Reverse Engineering Tools Production. *IEEE Transactions on Software Engineering*, Dec 1992.
- [GfC90] Judith E. Grass and Yih fam Chen. The C++ Information Abstractor. In *USENIX C++ Conference Proceedings*, pages 265–277, 1990.

- [gim] Gimpel Software. URL: <http://www.gimpel.com>.
- [Hol95] Jim Holmes. *Object-oriented Compiler Construction*. Prentice Hall, 1995.
- [Lip89] Stanley B. Lippman. *C++ Primer*. Addison-Wesley, Reading, MA, USA, 1989.
- [Mey92] Scott Meyers. *Effective C++ : 50 Specific Ways to Improve your Programs and Designs*. Addison-Wesley, Reading, MA, USA, 1992.
- [Mey96] Scott Meyers. *More Effective C++*. Addison-Wesley, Reading, MA, USA, 1996.
- [MK97] Scott Meyers and Martin Klaus. Examining C++ Program Analyzers. *Dr. Dobb's Journal*, Feb 1997.
- [ML91] Scott Meyers and Moises Lejter. Automatic detection of C++ programming Errors: Initial thoughts on lint++. Technical Report CS-91-51, Brown University, 1991.
- [par] Parasoft Corp. URL: <http://www.parasoft.com>.
- [R⁺91] J. Rumbaugh et al. *Object-oriented modeling and design*. Prentice Hall, 1991.
- [RL91] David S. Rosenblum and Alexander L. Wolf. Representing semantically analyzed C++ code with Reprise. In *USENIX C++ Conference Proceedings*, pages 119–134, April 1991.
- [SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1994.
- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, 2/ edition, 1991.
- [YNC90] Y.F.Chen, Michael Nishimoto, and C.V.Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, March 1990.